A few software engineering techniques for programming bioinformatics pipeplines

(git, bash, markdown, containers)

Philippe Veber

September 17th, 2025

Laboratoire de Biométrie et Biologie Évolutive

Motivations

Performing data analysis requires to write software (of a very specific kind but still).

Sofware engineering is a set of techniques to ease the production of software (in particular when working as a team) and make it of higher quality.

We'll see a few techniques/tools that are relevant when developing a data analysis pipepline

CAVEAT They do require some time to learn, but the effort is worth in the (not so) long term

Archiving past versions of my code

Motivations

When working on my project, it's safer to keep backups of my code, just in case I'd mess things up.

How about keeping (dated) copies somewhere?

- cumbersome (so I won't do it often)
- not easy to navigate/search/backtrack

Version Control is a much better alternative

We'll see later that archiving past versions of a code is fundamental to collaborative development and other advanced needs.

Version control

A version control program can take **snapshots** of a directory tree and helps navigating between those snapshots.

This is useful to:

- archiving/documentation of development steps
- backtracking to previous versions
- handling of multiple versions
- **collaborative development**

We'll use a program named `git`

Getting started

`git` can easily installed on Linux (e.g. in Debian/Ubuntu):

```
apt install git
```

but is also available on MacOS or Windows:

```
http://git-scm.com/download/mac
http://git-scm.com/download/win
```

All `git` commands follow the same pattern:

```
git SUBCMD ARGS*
```

Creating a repository

A repository is basically a directory equipped with a `.git` subdirectory, where `git` saves everthing it needs to.

Let's make our project directory a `git` repository:

```
$ cd camel-project
$ git init
Initialized empty Git repository in ~/git-m2-bee/camel-paper/.git/
$ tree -a
.
|-- .git
|   |-- branches
|   |-- config
|   |-- description
|   |-- HEAD
|   |-- [...]
|   `-- refs
|       |-- heads
|       `-- tags
`-- README.md

10 directories, 18 files
```

(My project is about the divergence time between camels and dromedaries)

Checking the state of the repository

Command : `git status`

```
$ git status
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
 README.md

nothing added to commit but untracked files present (use "git add" to track)
```

`git` tracks a file's history only if explicitly asked so, and warns us when there are untracked files in our repository.

Put a file under version control

Command : `git add FILE*`

This command tells `git` it needs to track the modifications of a given file from now on. It is then said under version control.

It needs to be done only once (well we'll see more about this later).

```
$ git add README.md
$ git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
 new file:   README.md
```

Take a snapshot

Command : `git commit -a -m MSG`

- A snapshot in `git` is called a **commit**
- The `-m` option introduces a message describing the changes in the new snapshot.

```
$ git commit -a -m "Started README"
[main (root-commit) fdafb43] Started README
 1 file changed, 3 insertions(+)
 create mode 100644 README.md
$ git status
On branch main
nothing to commit, working tree clean
```

Update, check, commit

Let's make a modification on our `README.md` and see what `git` says:

```
$ git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
 modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
$ git diff
diff --git a/README.md b/README.md
index e32739a..08fabba 100644
--- a/README.md
+++ b/README.md
@@ -1,3 +1,6 @@
 # On the divergence time between camels and dromaderies

-And how many humps did their common ancestor have?
+In this work, we propose an estimation of the divergence time between
+camels and dromaderies from genomic sequences. We also try to answer
+the difficult but major question in the field: how many humps did
+their common ancestor have?
$ git commit -a -m "rephrased README introduction"
[main 634e5ac] rephrased README introduction
 1 file changed, 4 insertions(+), 1 deletion(-)
```

Showing differences

To see which files have changed, `git status`

To see how those files have changed, use `git diff`

- lines prefixed with - are from the previous commit
- lines prefixed with + are from the current (uncommitted) state

Note: this is not something you can achieve easily with home-baked backups

Exercise

1. update your `README.md` file with more information
2. inspect the modifications with the commands `status` and `diff`
3. make a new commit (remember, no need to add the file again)
4. repeat this until you get the gist of it, maybe adding a new file to your project, like a `LICENSE` file

Print commit history

Command : `git log`

```
commit 634e5ace161fd480a1ec5e0d5308714f19ebc69f (HEAD -> main)
Author: Philippe Veber <philippe.veber@gmail.com>
Date:   Wed Nov 15 04:15:23 2023 +0100

    rephrased README introduction

commit fdafb431bcda725398236a9dc4c02458d391efa4
Author: Philippe Veber <philippe.veber@gmail.com>
Date:   Wed Nov 15 03:52:15 2023 +0100

    Started README
```

Print an old version of a file

Command : `git show COMMIT_ID:PATH`

- retrieve commit id using `git log`
- only the 6-8 letters of the id are generally enough

```
$ git show fdafb43:README.md
# On the divergence time between camels and dromaderies

And how many humps did their common ancestor have?
```

Note: use completion after the colon to navigate inside the (ancient) tree

Survival kit

Here are the basic commands to archive your work in a `git` repo:

Initialize a new (empty) `git` repo	`git init`
Put files under version control	`git add file1 file2 ...`
Take a snapshot	`git commit -a -m "commit descr"`
List uncommited files	`git status`
Print uncommited modifications	`git diff`
See commit history	`git log`
See previous version of a file	`git show aazef345:src/foo.c`

Daily routine

When starting to work on your project

1. `git status` to recall where you left things
    - if the repo is clean, then go to 2
    - else, is it a good time to commit?
        - if yes, then commit
        - if no, let's resume coding
2. work until it'd be nice to commit
3. commit with a nice message (I mean it, it's incredibly useful)
4. at the end of the day, push your commits (we'll see in the next section what this means)

Remote git repositories

Motivations

So far we have used `git` locally, which is already useful.

But `git` is also able to exchange data with repositories located on remote machines (aka **remote repositories**)

This is extremely useful:

- to have safer backups
- to collaborate on a project with other people

How to create a remote repository?

Simplest option: use a `git` hosting service

- machines on the internet that provide safe storare of git repositories, reachable via SSH
- plus a web site that offer many useful functionalities
  - creation/configuration/deletion of repos
  - graphical interface to explore a repo's contents
  - issue system to report bugs, discuss changes
  - wiki
  - continuous integration systems

In hosting services, a project is made of a `git` repo plus extra information related to those services.

What hosting service should I choose?

There are now several academic hosting services, based on an open-source software called **Gitlab**:

- UCBL
- CC IN2P3

Company-driven services like Github, Gitlab or Bitbucket propose free-of-charge (but limited) access, but beware of their business model that supposes some use of your code (see Github's Copilot).

Exercise

1. Create your account on some git hosting service
2. Add your SSH keys
   - avatar menu > Preferences, then left-panel menu > SSH keys
   - copy your **public** key (generally in `~/.ssh/id_rsa.pub`) in the form, and provide a name that will help you remember which machine it comes from
3. Create a repository for your project
   - go to the service's home page, > New project > Create blank project
   - in form, choose name and visibility
     - private means nobody can see the project except users you may specify later
     - public means anybody has READ-ONLY access
4. Use instructions to configure your local `git` repository to access your newly created repository
5. Enjoy how beautifully your `README` is now rendered on the web site
6. Create an issue using markdown syntax, references to other users (syntax `@login`) or commits (just the identifier)

Cloning a repo

Another means to proceed is to first create a project on a hosting service and then to **clone** its associated repository.

Command : `git clone URL [PATH]`

This command create an (independent) copy of an existing repo, with all its history. The optional path is for cloning into another local directory.

If a project is public, you can clone from it too! Generally there's a "Clone" button available on the root page of a project

- choose SSH address if you have permissions for this project
- HTTPS otherwise

Distributed Version Control

There is no difference in nature between a repo and its clone, and none is in some sense "more important" from `git`'s perspective.

This is called distributed version control. It lets contributors to a project organize the way they see fit.

Typically when several people work together, one copy is chosen to serve as a reference but it's pure convention.

After cloning a repo, the two copies are only related by the fact that the local one knows how to read/write on the remote one. Any two repos can be related in this way, even if they do not hold the same software project.

Exercise

1. make a clone repository from your project (beware, first go to another directory!)
2. try to clone a repo from some public project

Getting updates

Command : `git pull`

- will fetch commits from the cloned repo
- and update the working directory to latest commit

Sending updates

Command : `git push`

- will send commits to the cloned repo
- requires write access

Exercise

1. Make a commit and push it to your repo
2. Clone another group's repo, wait for them to make a commit and pull it
3. Give write permission to another group and ask them to push a commit

Survival kit

Here are the basic commands to access source code from a distant repo

Clone a distant repo	`git clone https://github.com/...`
Update local repo	`git pull`
Send commits to distant repo	`git push`

Branches

Motivations

Now that we can share repositories and push/pull commits between them, it has become a dangerous world.

How to deal with multiple people working on multiple aspects of a same project?

The answer provided by `git` relies on a fundamental concept: branches.

Commit

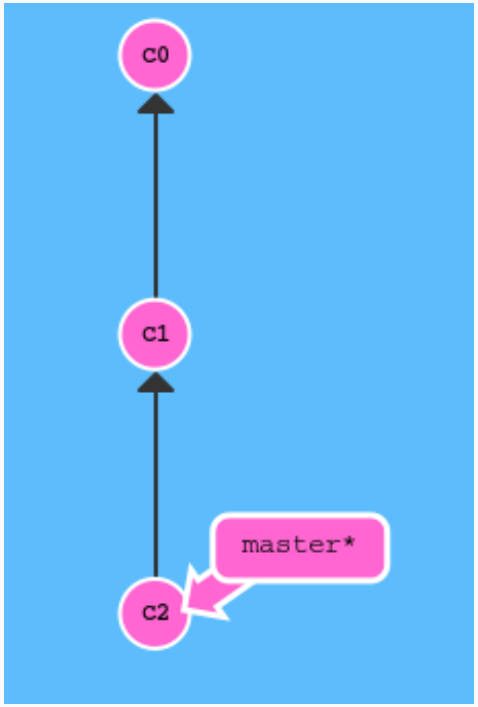
= snapshot of a directory tree. It is made of:

- a tree (= files with their path)
- author info, date
- description message
- a parent commit (previous commit in history)

All this information is (MD5) digested to produce the id of the commit.

Commit graph and branches

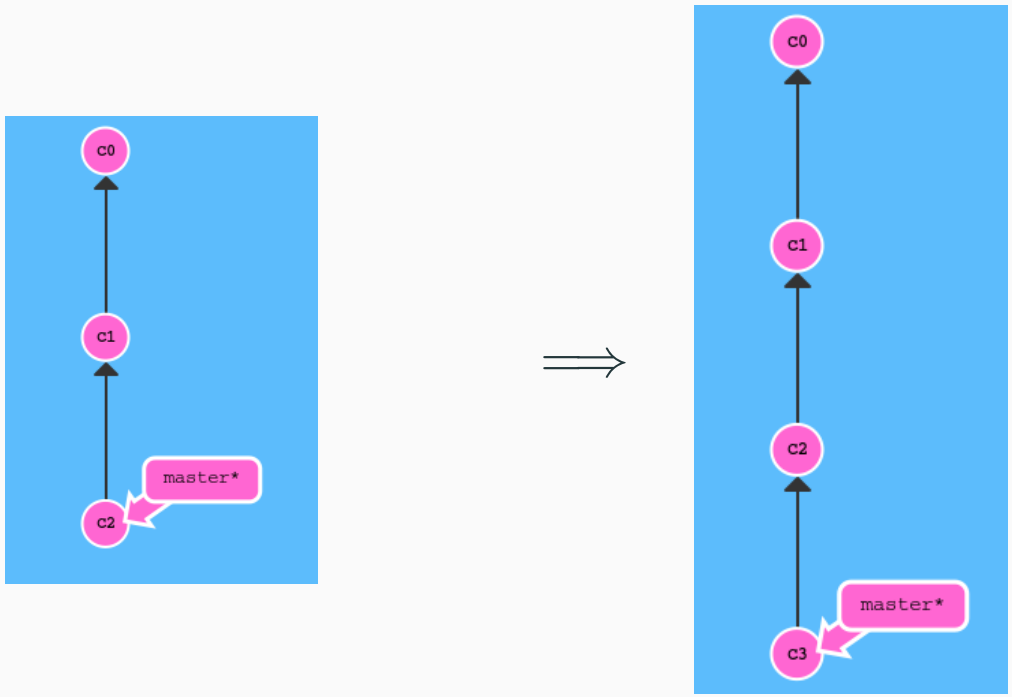
- commits form a graph
- each commit is linked to its parent(s)
- a **branch** is a pointer to a commit
- after repo init, there is a single branch called `main` (or `master`)
- at all time, one branch is selected as the "current branch"
- Demo



Effect of a git commit

When calling `git commit`

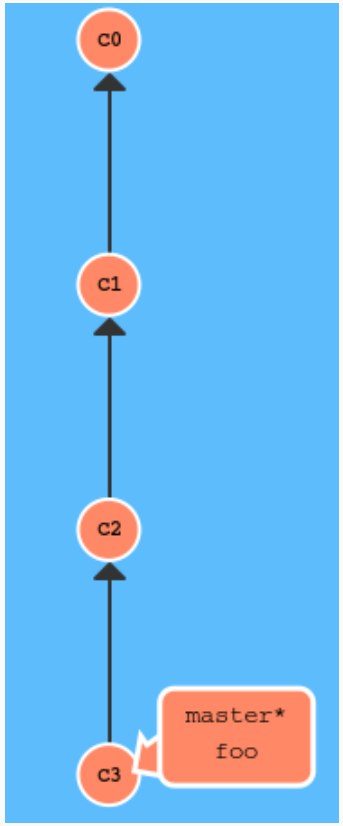
- a new commit is created in the graph
- the current branch is moved to the newly created commit



Creating a new branch

Command : `git branch BRANCH_NAME`

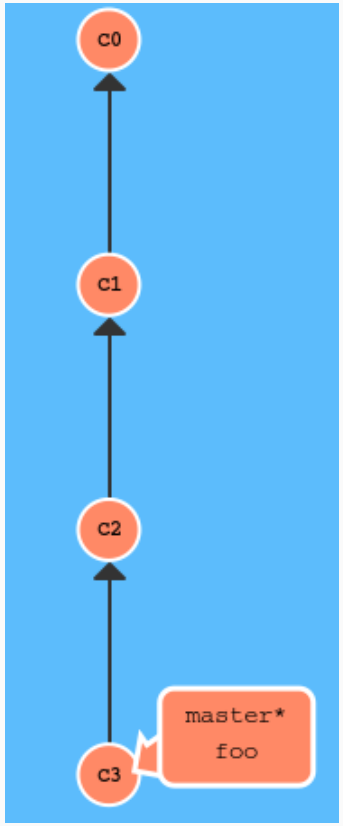
- branch is created at the location of the current branch
- very light operation (contrary to SVN)



Creating a new branch

Command : `git branch BRANCH_NAME`

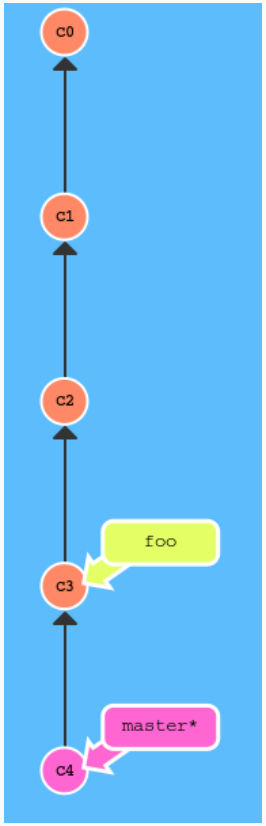
- branch is created at the location of the current branch
- very light operation (contrary to SVN)
- reminder: commit acts on current branch



Creating a new branch

Command : `git branch BRANCH_NAME`

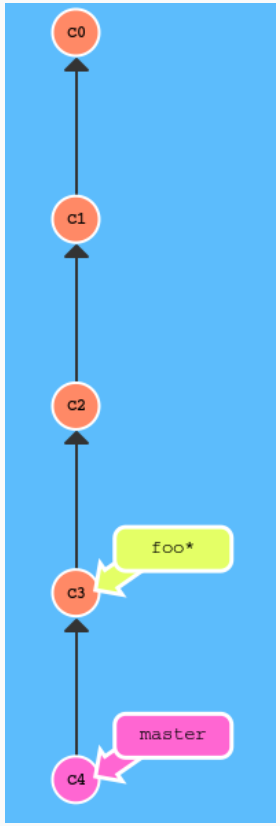
- branch is created at the location of the current branch
- very light operation (contrary to SVN)
- reminder: commit acts on current branch



Switching to some branch

Command : `git checkout BRANCH_NAME`

- Current branch is updated
- and so is the working directory, to reflect the state of the corresponding commit



Deleting a branch

Command : `git branch -d BRANCH_NAME`

- only the pointer is deleted, not the pointed commits
- but commit may be hard to find after branch deletion...

Practical uses of branches

- the main feature of branches is to allow switching between different versions very easily
- a branch should correspond to a topic, e.g. a bug fix, new feature, new release...
- especially useful when unsure when/if some modifications will be included (that is, very often)
- it permits several concurrent development processes

Typical scenario

Here's how we'd handle the repo after receiving a bug report

```
# user reports a segfault on my program

# create a new dedicated branch
git checkout -b fix-segfault              # equivalent to:
                                          #   git branch fix-segfault
                                          #   git checkout fix-segfault

# debug, debug, debug and:
git commit -a -m "fixed segfault caused by ..."

# finally, merge into master
git checkout master
git merge fix-segfault
```

Merging two commits

Command : `git merge BRANCH_NAME`

- `git` will try to merge two snapshots
  - the one pointed by current branch
  - the one pointed by the branch given in argument
- create a new commit
- move the current branch to the newly created commit
- the newly created commit has two parents!

Exercise

1. Create a branch and a few commits on it
2. Go back to your main branch and create another commit
3. follow the steps described in this page to get a beautiful log
4. remind me to lecture you about commit messages

Commit message

- title $<$ 60-70 characters
- OBJECT: ACTION
  - e.g. `intro: more details on genes related to hump growth`
- if necessary (often) skip two lines and more details
  - no more than 80 characters per line
  - use markdown syntax
  - use references to issues (e.g. #34)
  - and refererecences to commits (just put prefix of hash)

Example:

```
opam: add version constraint on camlzip

We require the ‘zip‘ findlib package, which is not provided by camlzip
1.04 (as mentioned in #144). We could instead require ‘camlzip‘, which
would make us compatible with all versions of camlzip. However, it
seems 1.05 is deprecating the ‘camlzip‘ findlib package name in favor
of ‘zip‘ (because camlzip is defined as an alias to zip, the library
file is zip.cma). Thus, might as well go with that.
```

Working with multiple repositories

Remote repos

A repo has a list of known remote repositories. Each remote repo has a name and a URL. The list can be printed with:

```
git remote -v
```

To add a new repo:

```
git remote add NAME URL
```

When cloning a repo, it is added as `origin`:

```
$ git clone https://github.com/ocaml/ocaml.git
[...]
$ git remote -v
origin https://github.com/ocaml/ocaml.git (fetch)
origin https://github.com/ocaml/ocaml.git (push)
```

Get updates from remote branch

To update the information available on a remote repo:

`git fetch REPO_NAME`

A branch of a remote repo is denoted REMOTE/BRANCH, e.g. `origin/master`. Remote branches can be used in any `git` operation:

```
$ git fetch origin                    # this is equivalent to git pull
$ git merge origin/master
```

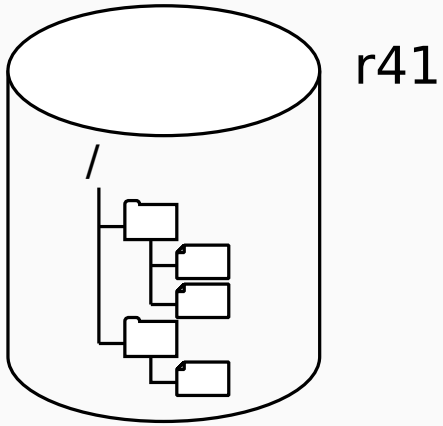
Sending commits to a remote branch

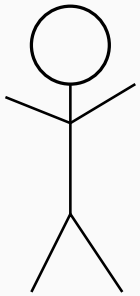
Command : `git push REMOTE BRANCH`

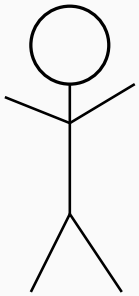
- e.g. `git push upstream v0.4.2`
- possible only if write access

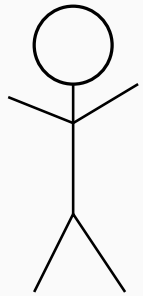
Dealing with conflicts

Typical scenario

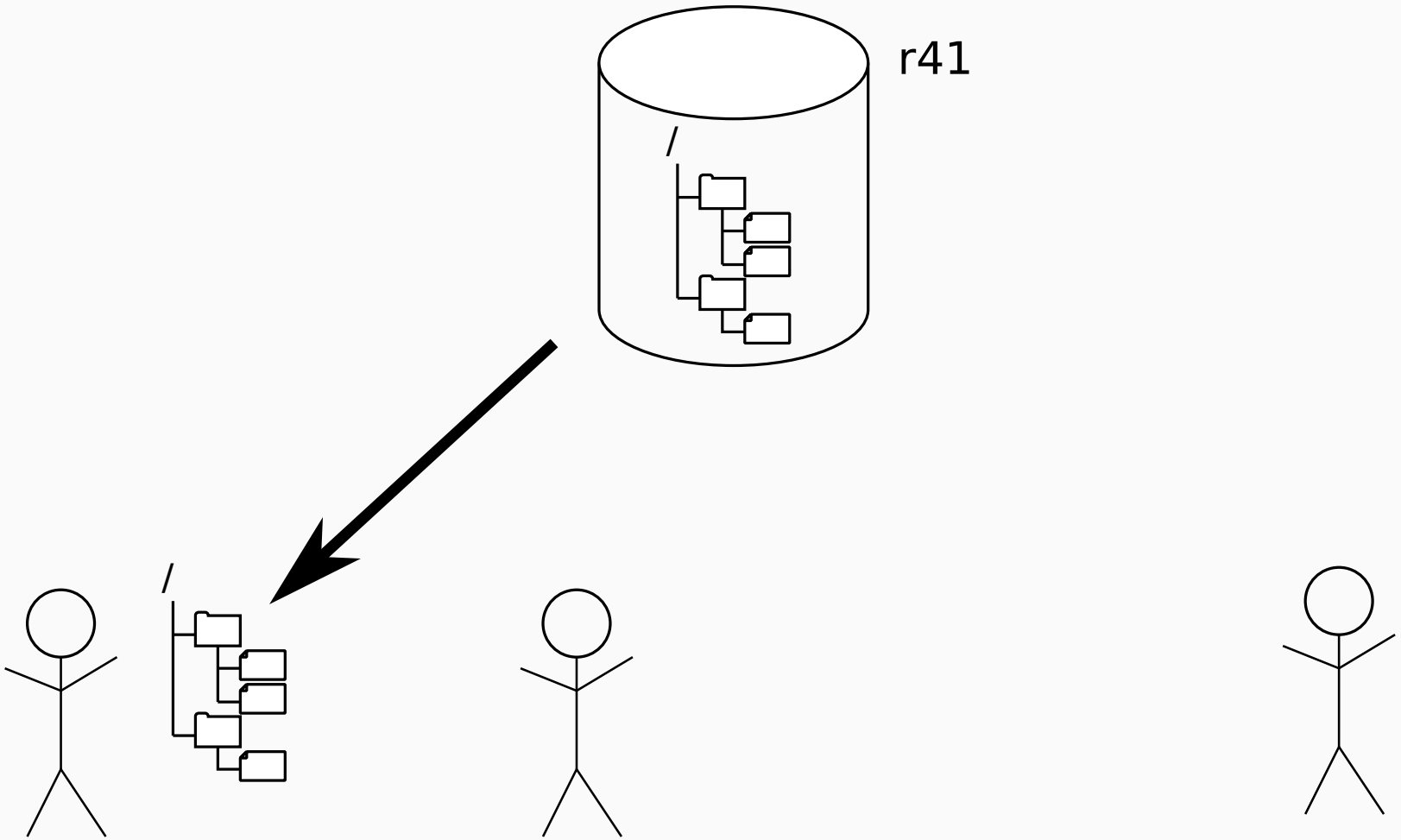




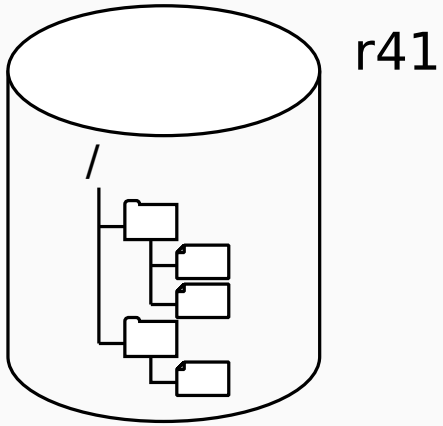


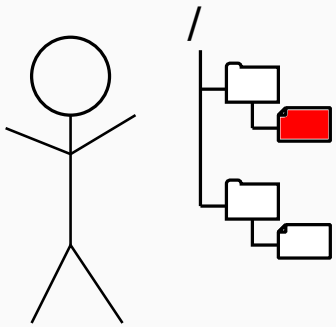


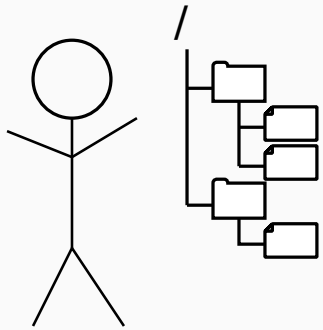
Typical scenario

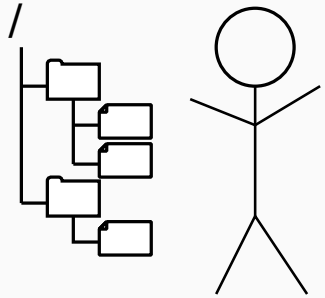


Typical scenario

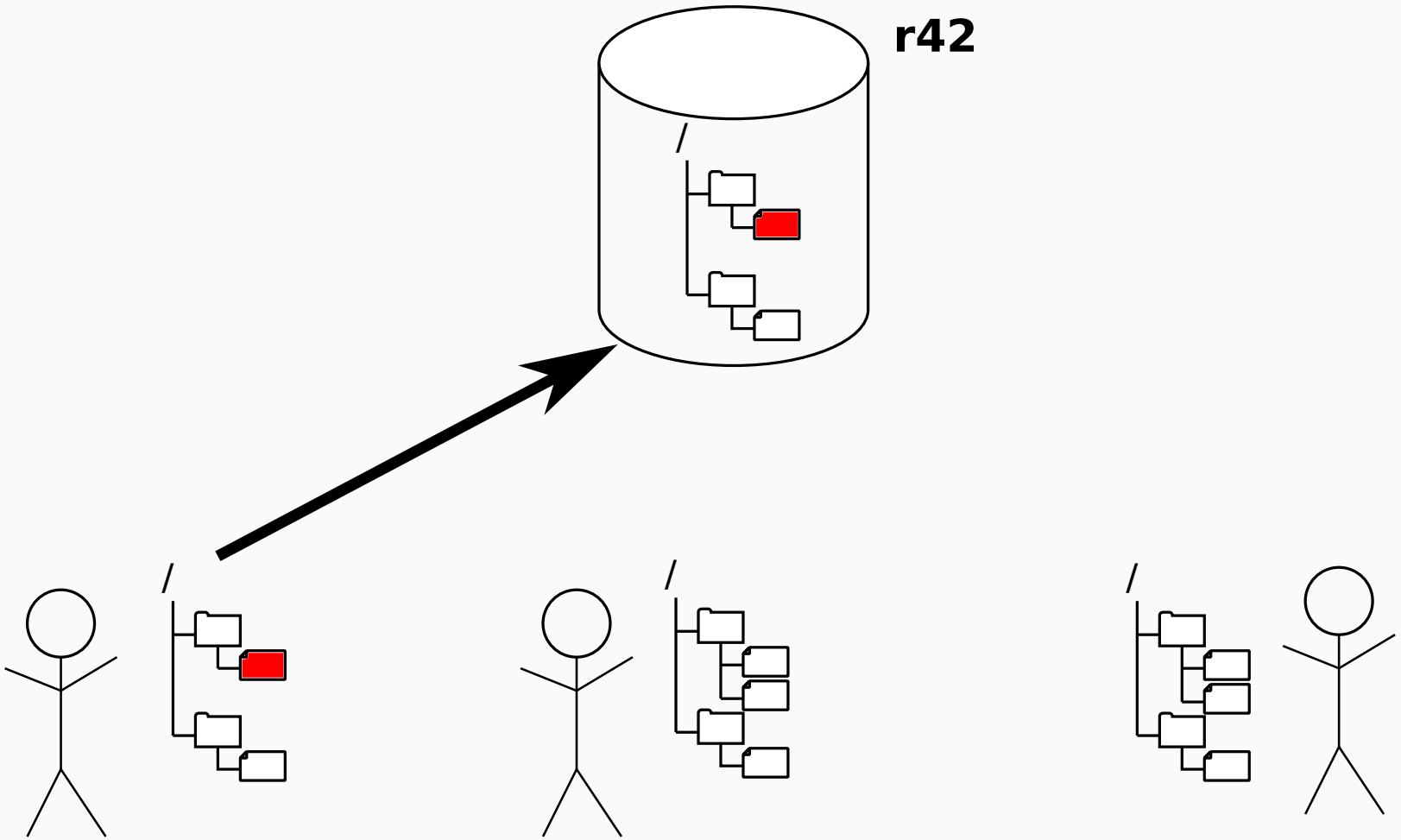




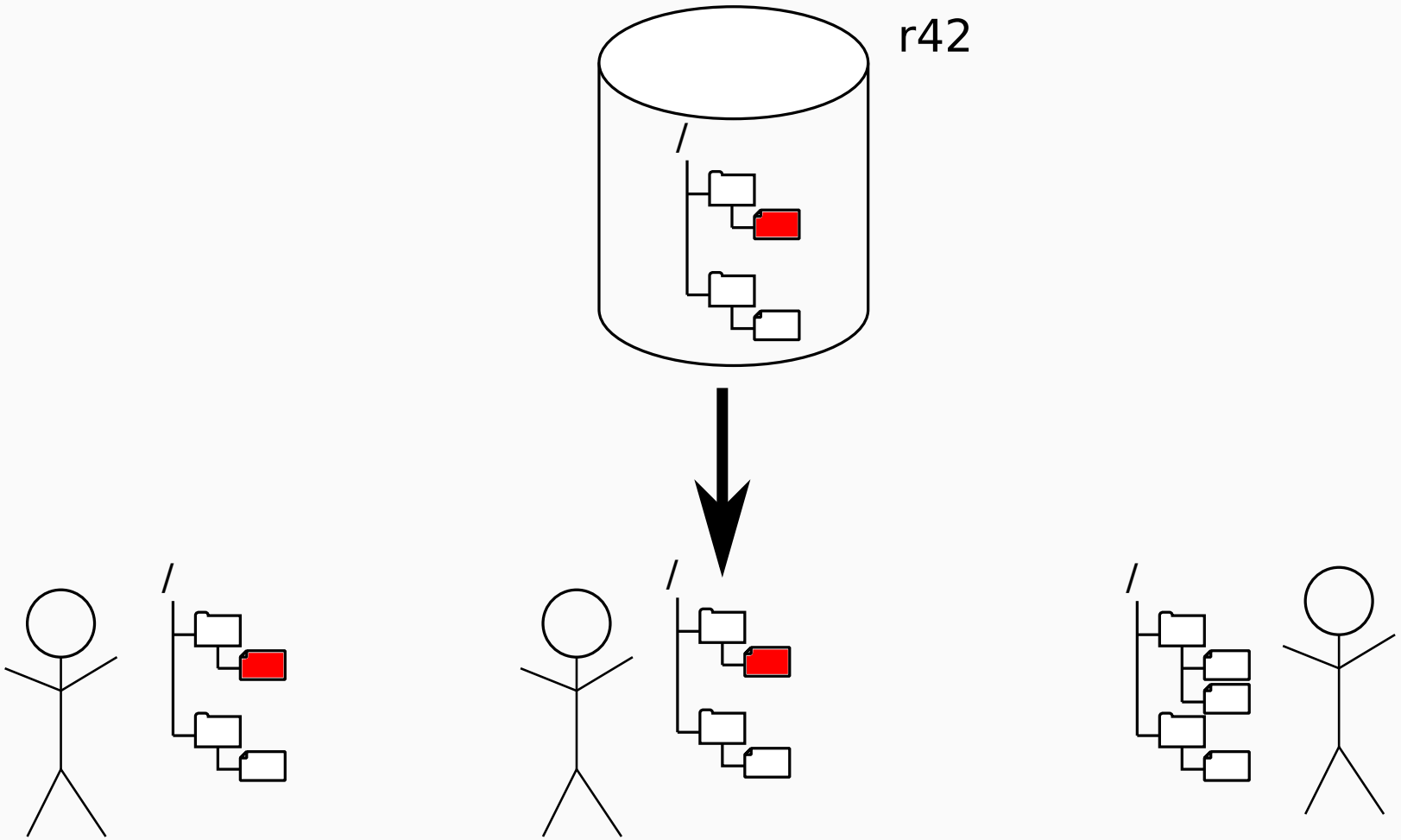




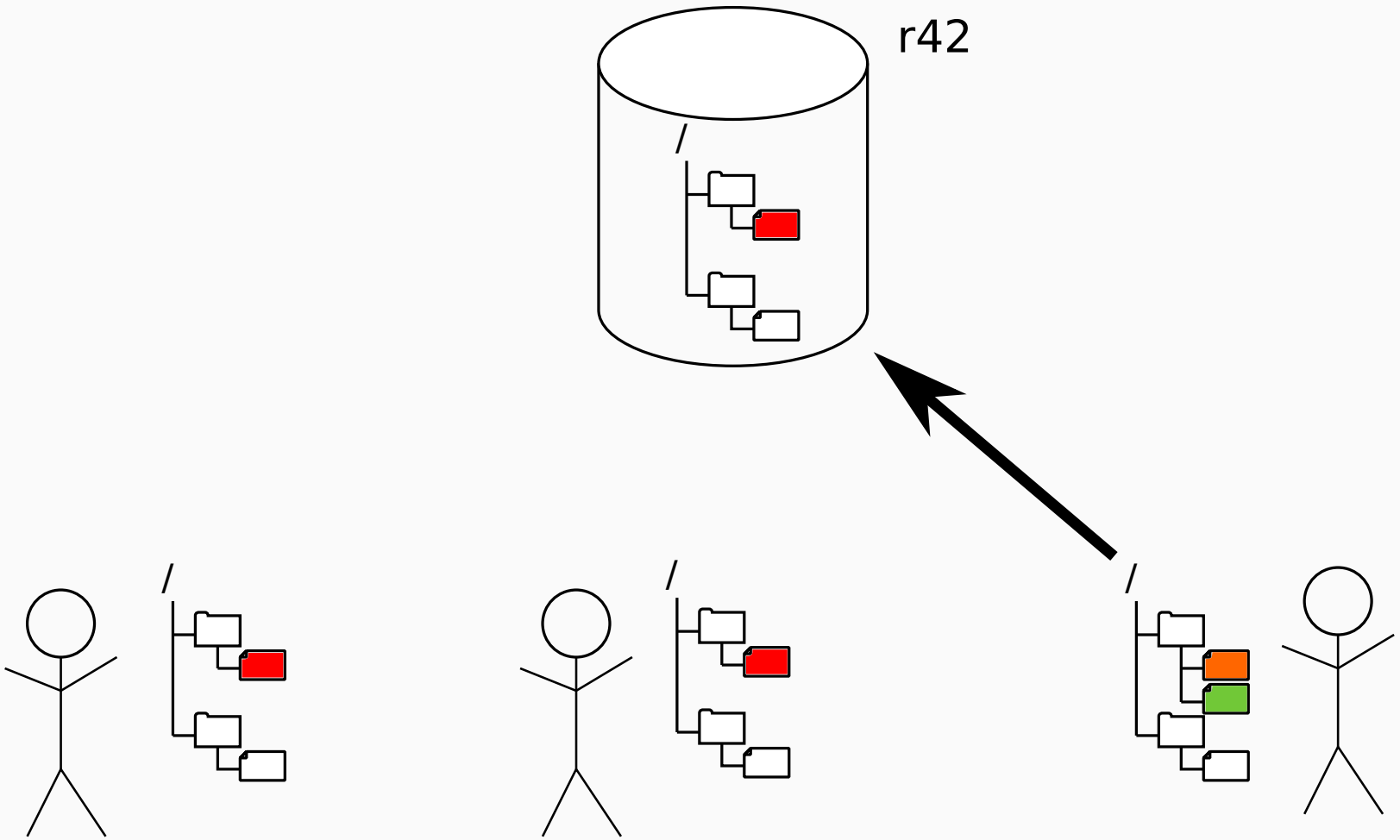
Typical scenario



Typical scenario



Typical scenario



What happens in practice

```
$ git push origin master
To /pandata/pveber/camel-project
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to '/pandata/pveber/camel-project'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

$\Longrightarrow$ Somebody has pushed to `origin/master` before me!

"Que faire ?" (Lénine, 1902)

- Update your view of the remote repo

```
git fetch origin
```

- Look at commit history

```
git log --all --graph --abbrev-commit
```

- Observe differences

```
git diff origin/master master
```

- Merge!

```
git merge origin/master
```

Let's try

```
$ git fetch origin
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From /pandata/pveber/camel-project
   7408985..4bf8768  master     -> origin/master

$ git log --all --graph --abbrev-commit
* commit 4bf8768
| Author: pveber <philippe.veber@gmail.com>
| Date:   Wed Nov 9 08:33:00 2016 +0100
|
|     Sections of the article
|
| * commit 69d96ef
|/  Author: pveber <philippe.veber@gmail.com>
|   Date:   Wed Nov 9 08:32:11 2016 +0100
|
|       wrote plan of the article
|
* commit 7408985
| Author: pveber <philippe.veber@gmail.com>
| Date:   Tue Nov 8 23:21:02 2016 +0100
|
|     Expand introduction
```

Let's try

```
$ git diff origin/master master
diff --git a/paper.md b/paper.md
index 4e6d868..d94ab2d 100644
--- a/paper.md
+++ b/paper.md
@@ -6,8 +6,8 @@ while the latter has one only. The goal of the current paper is to
 estimate the divergence time between the two species and infer the
 number of humps of the common ancestor.

+# Methods
+
 # Results

 # Discussion
-
-# Methods

$ git merge origin/master
Auto-merging paper.md
CONFLICT (content): Merge conflict in paper.md
Automatic merge failed; fix conflicts and then commit the result.
```

Conflict

Definition
Two commits are in conflict when they contain different updates to the same area of a file.

- purely syntactic notion: `git` knows nothing about the semantic of your files
- $\Rightarrow$ conflicts are to be resolved manually
- absence of conflicts doesn't mean the merge is successful
- two commits can modify files in incompatible way (i.e. such that the merge doesn't compile or work properly)

Resolving conflics: list files with conflicts

```
$ git status
On branch master
Your branch and 'origin/master' have diverged,
and have 1 and 1 different commit each, respectively.
  (use "git pull" to merge the remote branch into yours)
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

 both modified:   paper.md

no changes added to commit (use "git add" and/or "git commit -a")
```

Resolving conflics: visualize conflicts

```
$ cat paper.md
# Introduction

Camels and dromedaries are animals with humps that live in dry
lands. Although they look very similar, the former have two humps
while the latter has one only. The goal of the current paper is to
estimate the divergence time between the two species and infer the
number of humps of the common ancestor.

<<<<<<< HEAD
# Methods

# Results

# Discussion
=======
# Results

# Discussion

# Methods
>>>>>>> origin/master
```

Resolving conflics: solve conflict

```
$ emacs paper.md
$ cat paper.md
# Introduction

Camels and dromedaries are animals with humps that live in dry
lands. Although they look very similar, the former have two humps
while the latter has one only. The goal of the current paper is to
estimate the divergence time between the two species and infer the
number of humps of the common ancestor.

# Methods

# Results

# Discussion

$ git add paper.md

$ git commit
[master a0aa9e2] Merge remote-tracking branch 'origin/master'
```

Post-conflict history

```
$ git log --all --graph --abbrev-commit

*   commit a0aa9e2
|\  Merge: 69d96ef 4bf8768
| | Author: pveber <philippe.veber@gmail.com>
| | Date:   Wed Nov 9 09:11:45 2016 +0100
| |
| |     Merge remote-tracking branch 'origin/master'
| |
| * commit 4bf8768
| | Author: pveber <philippe.veber@gmail.com>
| | Date:   Wed Nov 9 08:33:00 2016 +0100
| |
| |     Sections of the article
| |
* | commit 69d96ef
|/  Author: pveber <philippe.veber@gmail.com>
|   Date:   Wed Nov 9 08:32:11 2016 +0100
|
|       wrote plan of the article
|
* commit 7408985
| Author: pveber <philippe.veber@gmail.com>
| Date:   Tue Nov 8 23:21:02 2016 +0100
|
|     Expand introduction
```

A word of advice on conflicts

- update your branch before you start working on something
- make frequent and small commits
- if you work on a branch, rebase it as often as possible (more on this later)

Exercise

By groups of two, create a situation of conflict and fix it!

Using markdown for documentation

What is Markdown

Markdown is a syntax to describe documents

- text file with annotations to express typography (italic/bold type, titles, lists *etc*)
- markdown documents should have a `md` extension.
- there's a **compilation** step to produce a PDF or HTML file
- excellent compromise btw
  - minimal typing
  - readability without compilation
  - expressivness for typography

Syntax sample

````
# Document title
## First section
Text can be made *italic*, **bold** or ‘monospace‘. It’s easy
to create lists:
* one
* two

Or to add code:
‘‘‘R
f <- function(x) {
  x + 1
}
‘‘‘
and even formulas like $\sum_{i=1}^n i^2$ or images:
![alt text for screen readers](/path/to/image.png "Text on mouseover")
````

See a more complete document

Compilation

One way to obtain PDF or HTML files from markdown documents is to use pandoc

A typical invocation:

```
$ pandoc --from=markdown --to=latex --output=doc.pdf doc.md
```

Exercise

Let's pretend we start a data analysis project:

1. create a directory somewhere
2. open a `README.md` file in it using your favorite editor
3. use various markdown syntax elements to describe your project
4. (optional) try to compile it using `pandoc`

How to organize a data analysis project?

A simple structure

1. `README.md` a markdown file with:
   - a short summary of the project
   - a short list of instructions on how to install the required dependencies and run the analysis
   - general info (list of involved contributors and research institutions, associated bibliographic references)
2. `LICENSE` file
   - you can use CeCILL-B by default, but it's better to discuss the matter at the start of the project
3. `src` a directory containing all code (can be organized by languages)
4. `data` a directory containing datasets **if and only if**
   - they're not too big (not more than 1MB)
   - their extraction required some non automatable process
5. `res` **the one and only** directory containing the results of your program

Sorry to insist

your programs should never write outside of `res`

Corollary: if a dataset can be obtained automatically, it goes to `res` not to `data`

Your goal

> Once you're finished with your project, it should be easy for anybody to reproduce all calculations from data to article figures by invoking a single command

Often difficult to achieve, but the closer you get from this the better

How to organize your programs

In programming you typically divide your tasks into sub-tasks until you arrive to easily manageable bits. The main conceptual tool for this are **functions**.

In data analysis, it is often that we need several programs that communicate through files:

- some calculation is performed by a call to an external executable
- another is so long that it's better to save its result on disk rather than to keep in memory
- a result is so large that it is not possible to keep it in memory
- we backup intermediate result so that, when developing, it's not necessary to run everything from the beginning

This form of programming is called *pipeline programming*

- functions $\rightarrow$ programs
- variables $\rightarrow$ files

Pipeline design

You need to think hard on how to organize the analysis in manageable steps, and the layout of your result directory.

One way is to draw a graph where some boxes represent intermediate results (and their location in `res`).

Another (which I prefer) is to use pseudo-code

In any case, prepare a (markdown) document to specify the layout of your result directory:

```
| Location             | Description
|:---------------------|:-----------------------------------
| genome/$ID.fa        | Genome sequence for species $ID retrieved from
| mapped_reads/$S.sam  | Mapped reads for sample $S using bowtie2
| ...                  | ...
```

This can go to a doc directory (not in `res` of course!)

Code organization

You'll often need to use several programming languages, it's better not to mix them:

```
.
|-- LICENSE
|-- README.md
|-- data
|-- res
`-- src
    |-- R
    |-- bash
    |-- ocaml
    `-- python
```

Choosing a programming language

For calculating purpose, choose whatever you want but:

- maximize your chances to produce a not-too-buggy program
  - achieving bug-free programs is really, really hard
  - choose a language that helps writing code that is **SIMPLE TO READ**
- and that has a performance profile well suited to your problem
  - compiled programs vs scripts
  - use optimized numerical libraries when possible
  - be suspicious as soon as something lasts more than 5 min

Programming requires a lot of experience and know-how, it is important you go discuss with people more knowledgeable than you.

Choosing a programming language

For running the different steps of the pipeline, there are several alternatives but start considering `bash`

- yes it might get painful for more advanced projects
- but simple things can be done simply
- everybody knows it
- it's reasonably portable and stable over time

When you start feeling `bash` is too limited for what you need, think hard to identify what is the problem and whether existing solutions bring more help than harm.

Bash programming tricks

Each script basically corresponds to a function

Look after the header of your scripts:

- a bash script should start with a shabang: `#!/bin/bash`
- then a multi-line comment explaining
  - what the script does, from a high-level perspective, then
  - how to call the script on the command-line
  - the assumptions on arguments
  - what are the outputs and where they are located

```
#!/bin/sh
## Description: does this and that
##
## Usage: myscript [options] GENOME
##
## Arguments:
##   GENOME        A fasta file
##
## Options:
```

A nice advanced trick to meditate on

```
#!/bin/sh
## Usage: myscript [options] ARG1
##
## Options:
##   -h, --help    Display this message.
##

usage() {
  [ "$*" ] && echo "$0: $*"
  sed -n '/^##/,/^$/s/^## \{0,1\}//p' "$0"
  exit 2
} 2>/dev/null

main() {
  while [ $# -gt 0 ]; do
    case $1 in
    (-h|--help) usage 2>&1;;
    (--) shift; break;;
    (-*) usage "$1: unknown option";;
    (*) break;;
    esac
  done
```

Bash programming tricks

These are **REALLY** important:

- Stop execution on error
  - add `set -e` after the header
- Assume constant working directory
  - using `cd` all over the place will only bring confusion, bugs and doom
  - if you really need to change directory, use this construct to add a clear scope

    ```
    (cd res/proteins && grep RAR list > filtered_list)
    ```
- note the use of `&&` in the previous example!

Exercise

Propose a design for a (simple) analysis of your choice

Dealing with dependencies

Motivations

The code written for a data analysis necessarily assumes some third-party programs/libraries are installed in the system

- for some the asumption is reasonable (e.g. systems calls for opening files)
- but for specific (scientific) needs, those `dependencies` will not be installed, or not with the correct version

How to make our code executable elsewhere?

Solution 1: thorough documentation

You can list in an `INSTALL` file all software your analysis rely on, with the exact version you used.

Spoiler alert, it does not work for most cases:

- you will forget something
- installing all deps could be too much effort or too skill-demanding for your audience
- it might be really technically challenging if one has incompatible constraints for the version of some library/program you use
- sometimes not possible at all if some installation requires root privileges on a shared computing infrastructure

This is better than nothing, but it does not really qualify as an answer to a reproducibility requirement

Solution 2: using containers

Linux has a mechanism that allows to run processes in isolation of the rest of the system, in a kind of box called a **container**.

In this container, processes don't see processes outside of the container, and have their own file system and network interfaces.

It is possible to exploit this mechanism to run programs without installing them. In practice one:

- downloads an image of a system where a given program is installed
- runs the program inside a container that sees a new file system containing the downloaded image

Solution 2: using containers

To use containers, there are two main programs known as `singularity` and `docker`. For computation on a shared infrastructure only `singularity` can be used.

Here's an example to use `bowtie2`, a tool for NGS data.

```
# Download an image for bowtie2
$ singularity pull \
  https://depot.galaxyproject.org/singularity/bowtie2:2.4.1--py37h4ef193e_2

# Run the executable inside a container
$ singularity exec ./bowtie2\:2.4.1--py37h4ef193e_2 bowtie2 --help
```

Solution 3: use source distributions

`bioconda` is a popular choice, but technically problematic. When available, prefer `guix` which offers near-perfect reproducibility.